Autoencoders and Image Generation

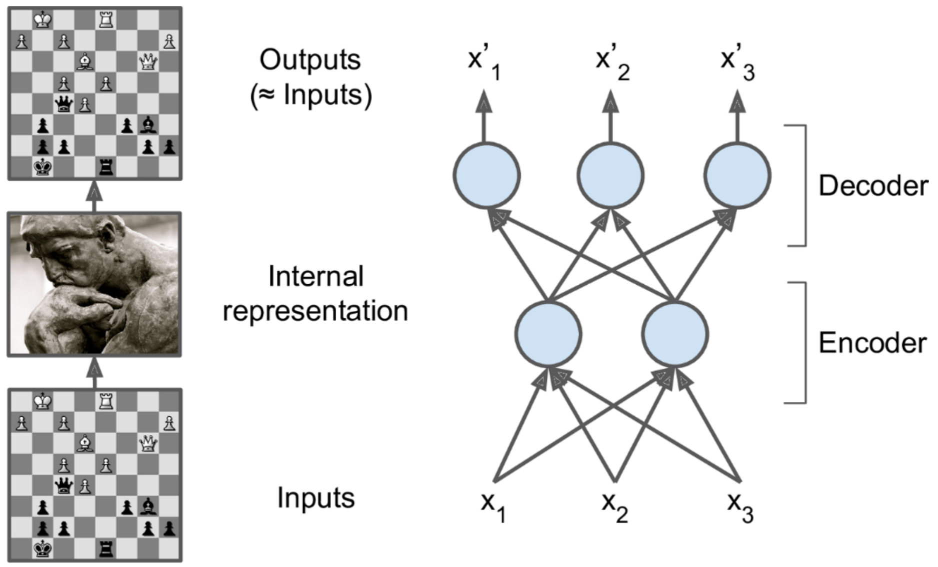
CMPUT 328

Nilanjan Ray

Picture source: “Hands on Machine Learning with Scikit-Learn & TensorFlow” by Aurelien Geron

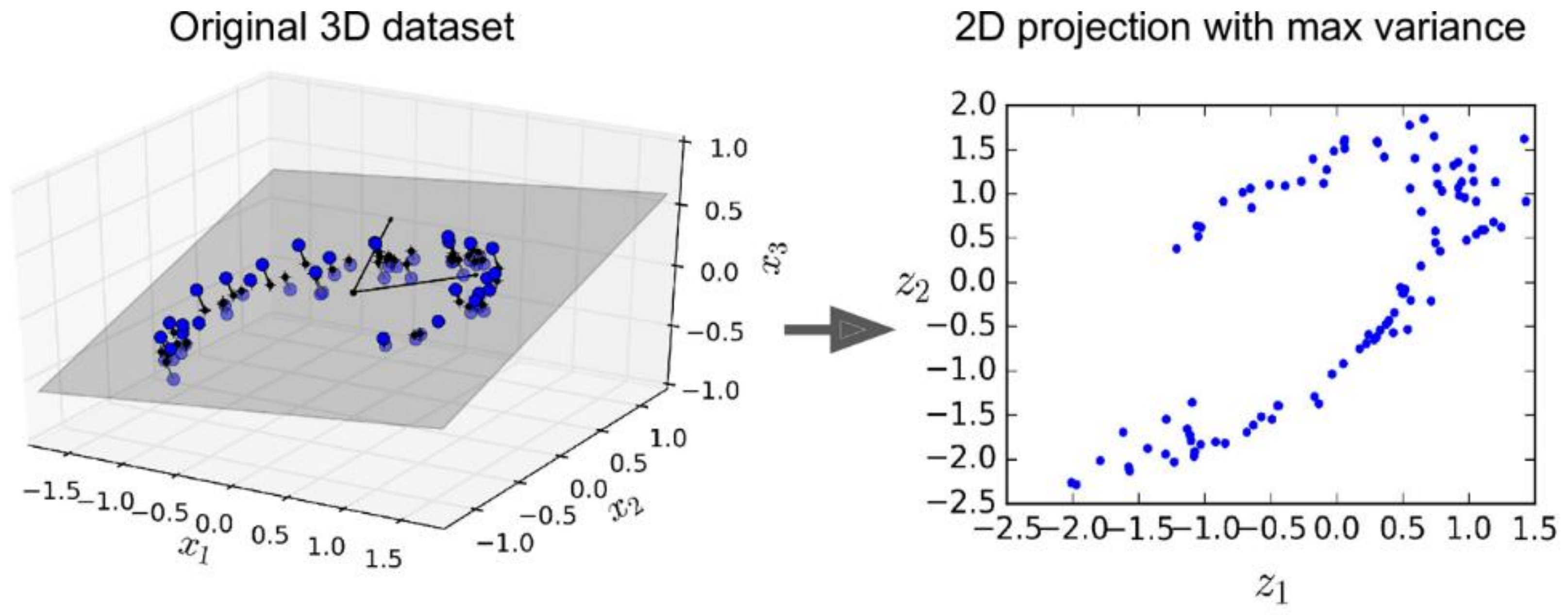
Part I: Autoencoders

Representation learning by autoencoders



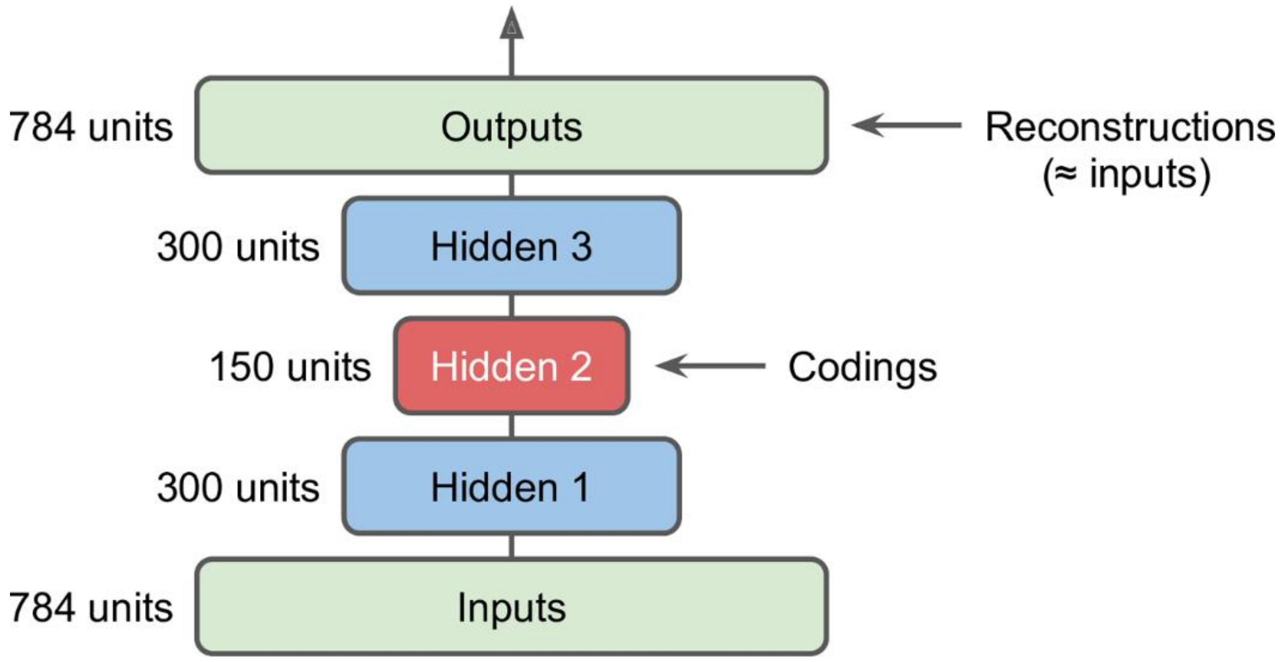
Can we learn interesting, hidden representations from unlabeled data?

Extract underlying (low) dimensionality



Even though the data points are 3D, they more or less lie on a 2D plane.

An example autoencoder for MNIST

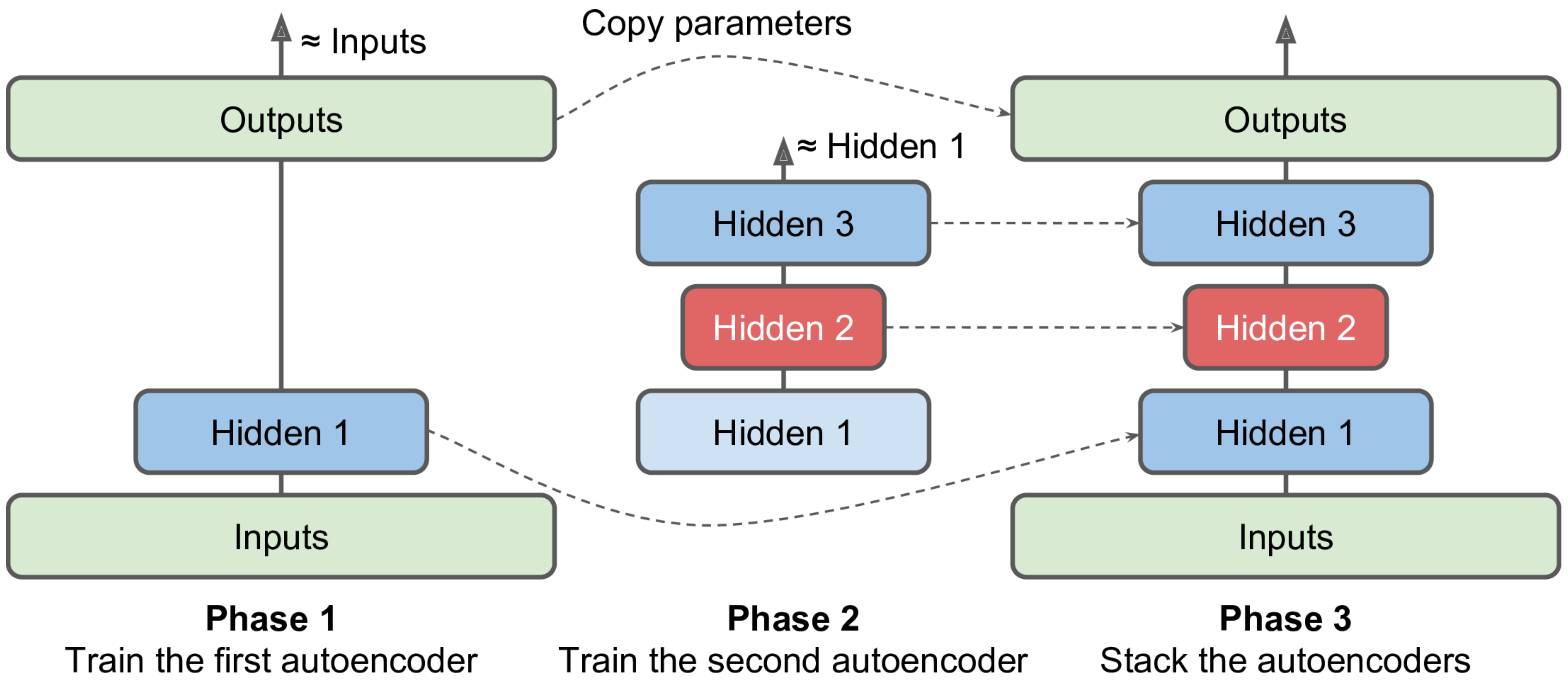


MNIST_AE.ipnyb implements this architecture

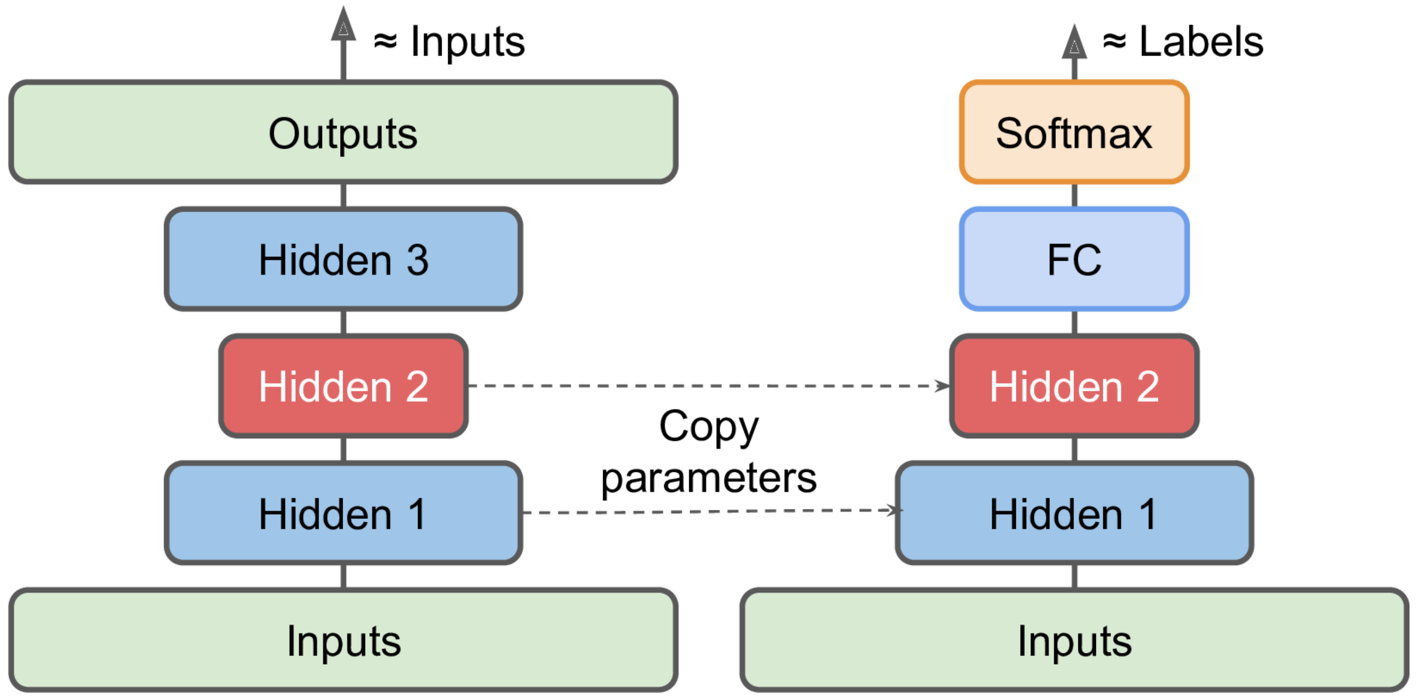
Convolutional AE

- We can easily replace fully connected layers by all convolutional layers with proper padding
- Look at MNIST_AE.ipnyb

Phased training in AE



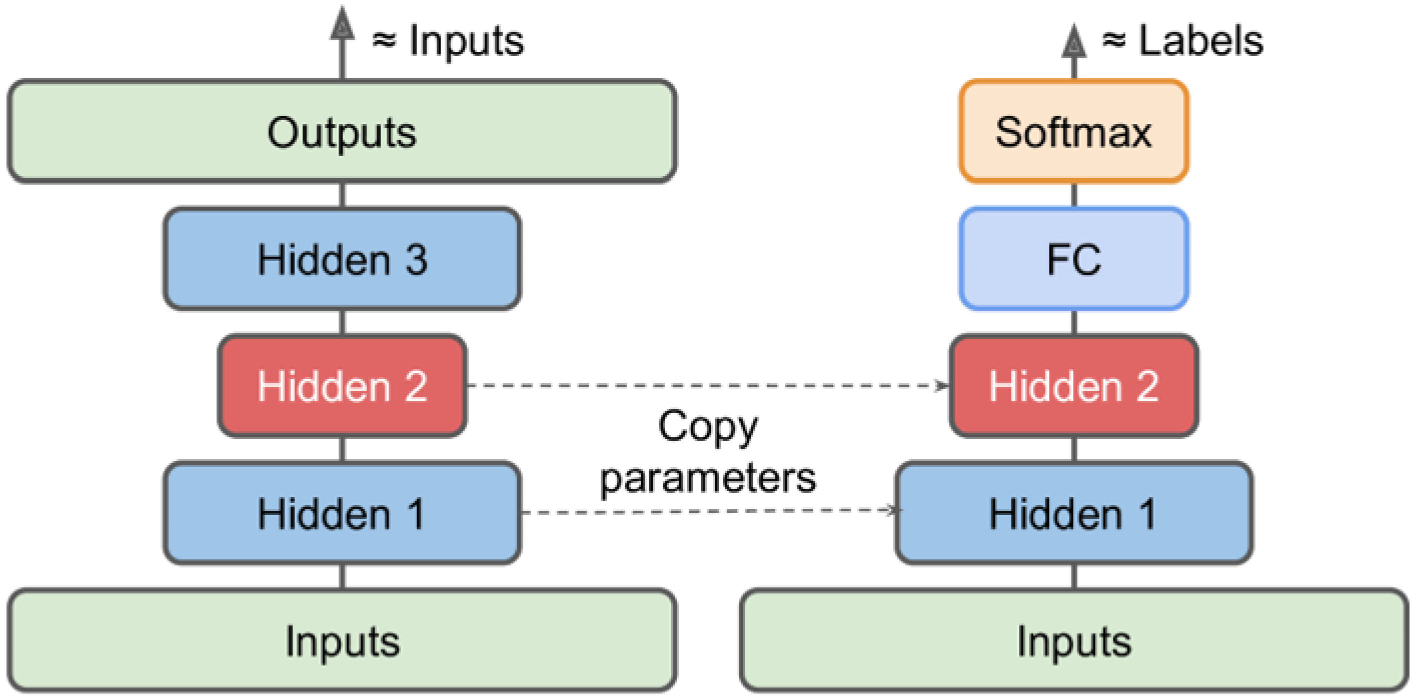
Classification from AE



Phase 1
Train the autoencoder
using all the data

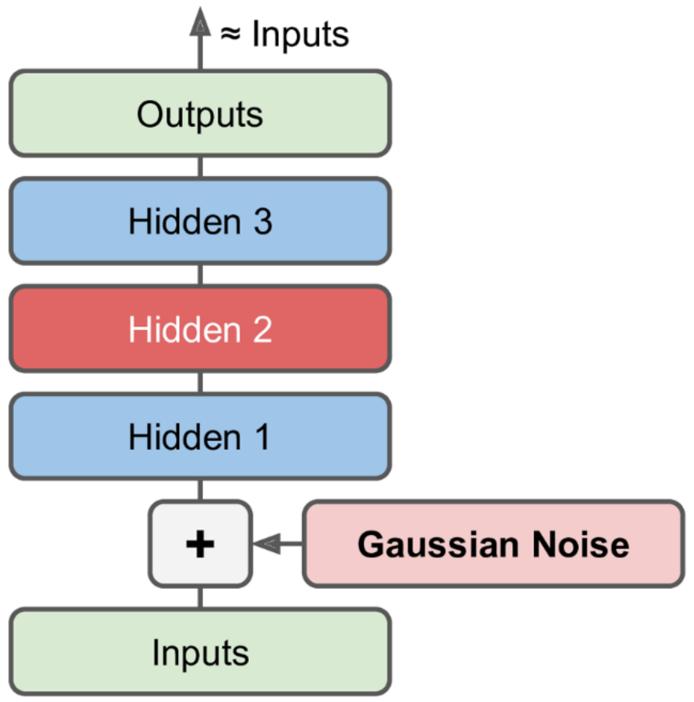
Phase 2
Train the classifier
on the *labeled* data

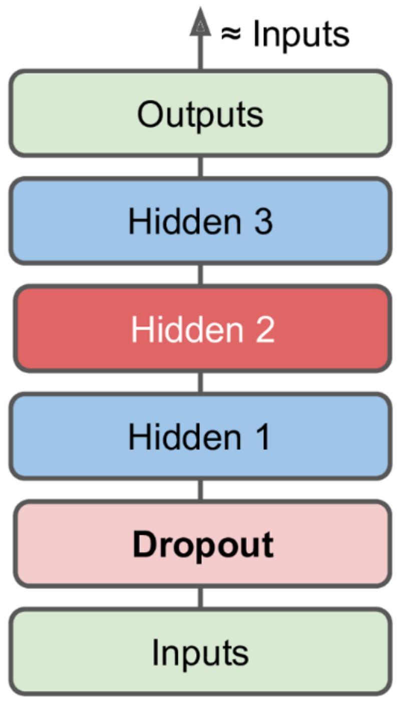
Semi-supervised learning



Often, labeled data is partially available: maybe 80% training data is unlabeled, only 20% is labeled. How do we make use of unlabeled data during training a classifier?

Denoising AE

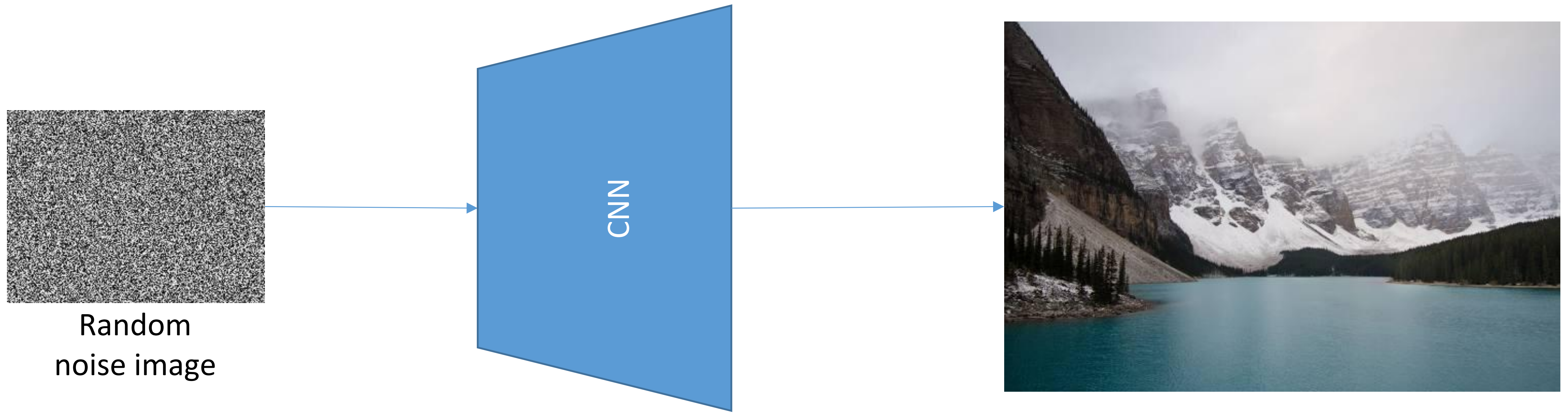




Add noise or insert a dropout layer

Part II: Image Generation

Generating images by neural networks



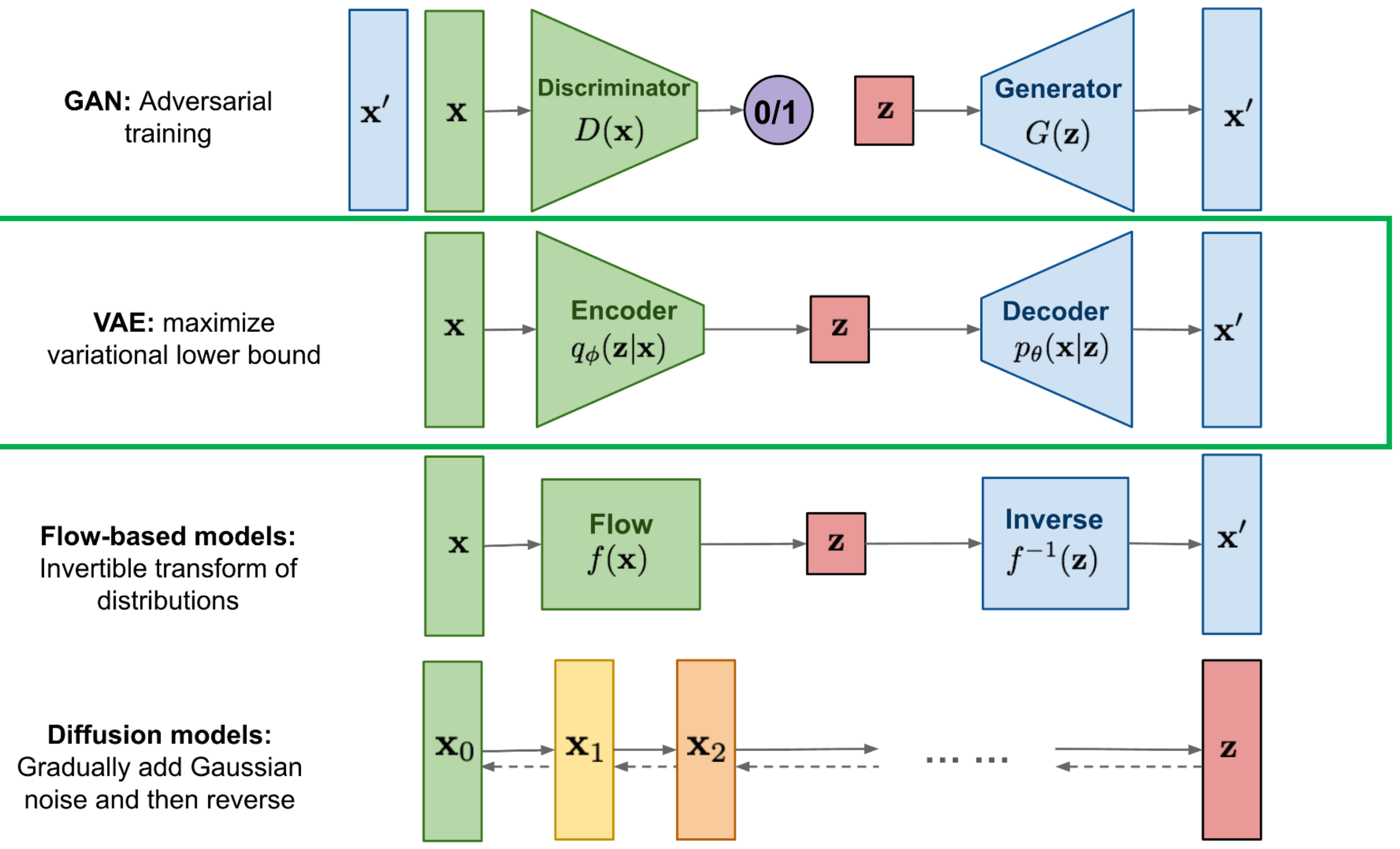
Random noise image

Picture

This is a typical setup: generate images with random noise inputs

Types of image generators

Today's topic



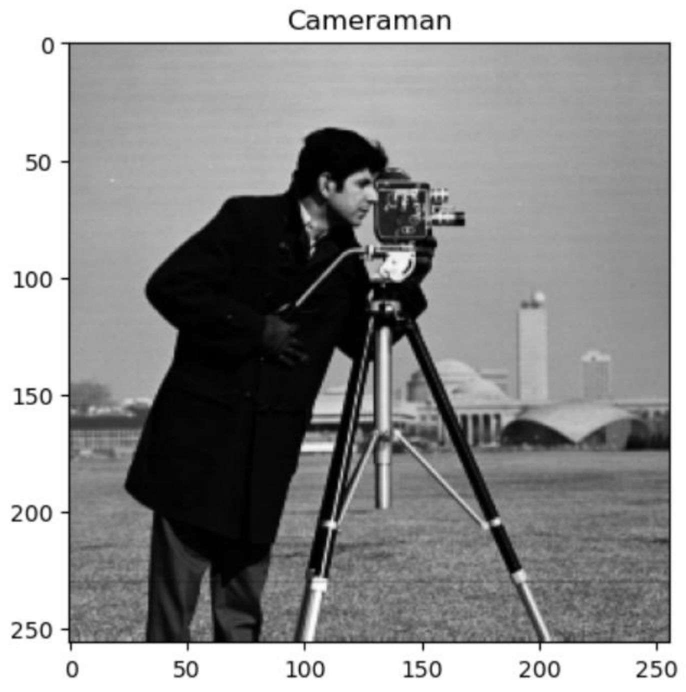
Picture: https://lilianweng.github.io/posts/2021-07-11-diffusion-models/

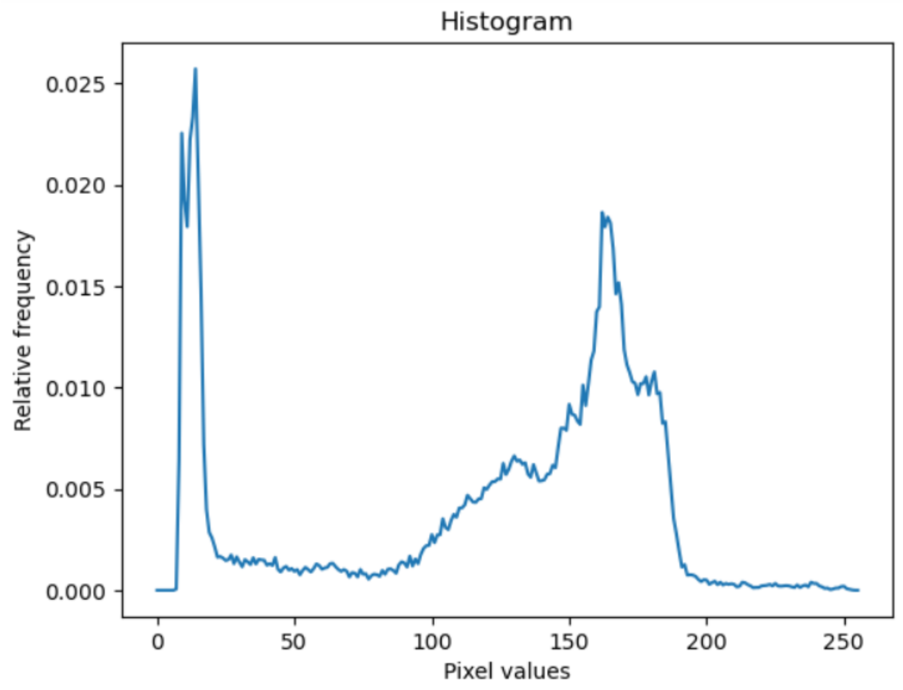
Probability primer

- We need to refresh our understanding of probability and random variables
- Gaussian probability density function
  - Scalar
  - Vector
- Generating random variables from a pdf
- Conditional density function

Probability mass function (pmf)

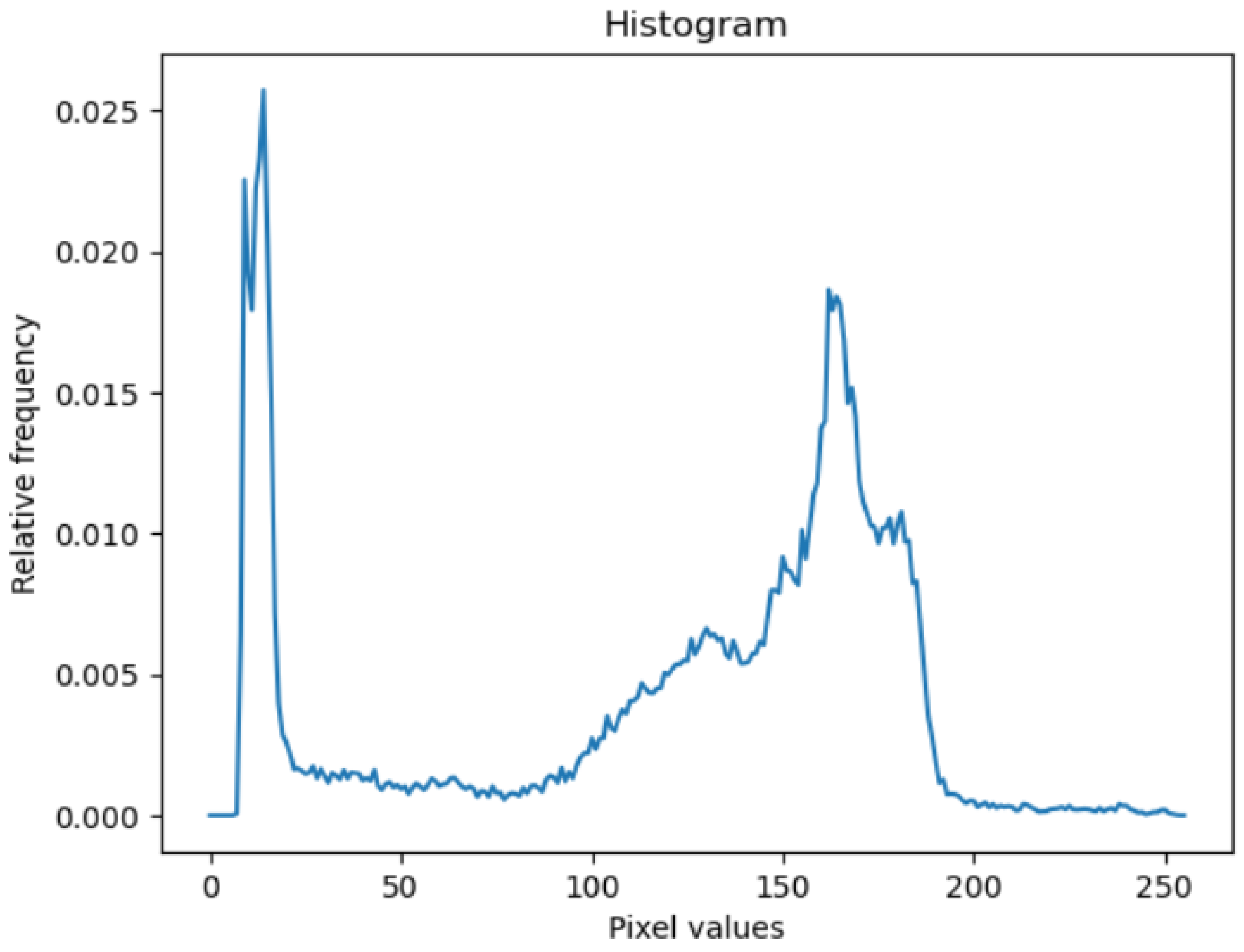
A familiar example: a grayscale image histogram





Random variables: Pixel values

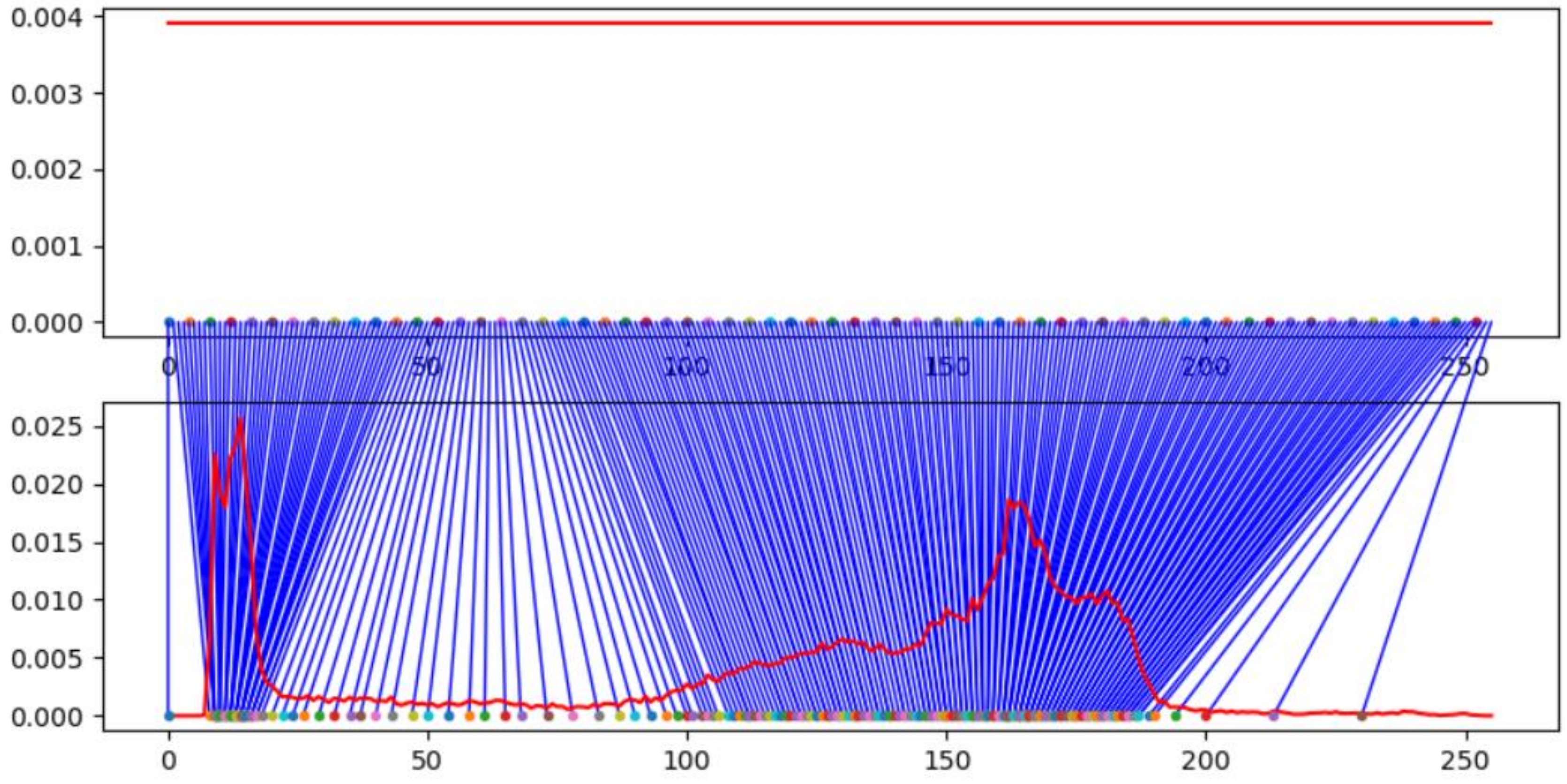
PMF = mass attached to random variables



Sampling random variables from a pmf

- There are various methods
- Here’s an example method:
- Generate random variable from a **familiar distribution**, such as a uniform distribution or a Gaussian distribution– these in-built functions are available to us
- Then transform this r.v. so that the transformed r.v. follows our target pmf
  - How?
- At the end of the day, it will become a function call (in numpy or pytorch)

Transformation of r. v. visually



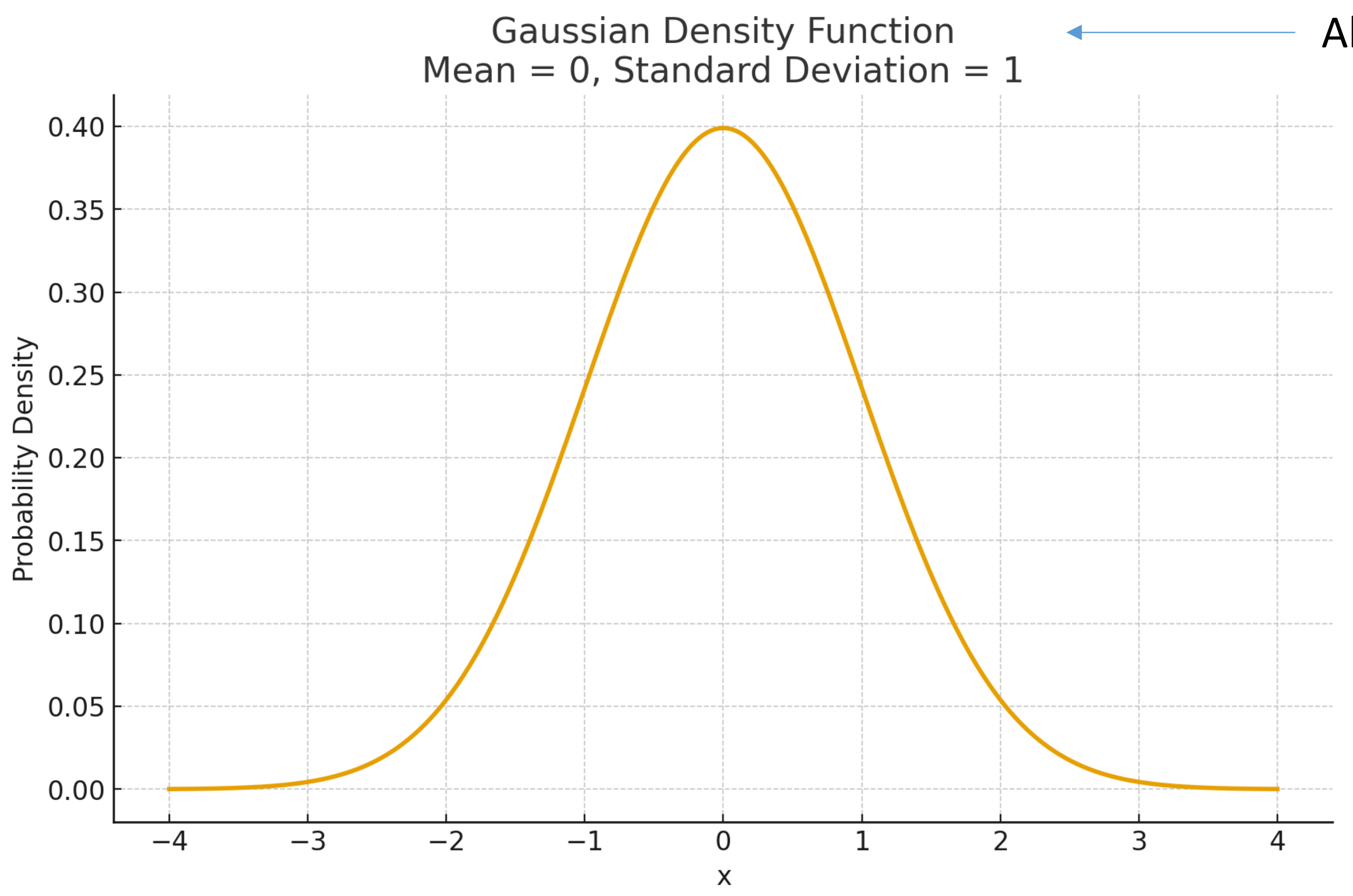
Uniform histogram

Cameraman histogram

Uniformly distributed r.v. (uniformly spaced points) are mapped (aka **transformed**) to r.v. that follows cameraman pmf

The original space is being **stretched** and **compressed** to mimic the mass distribution in cameraman pmf.

Gaussian density function

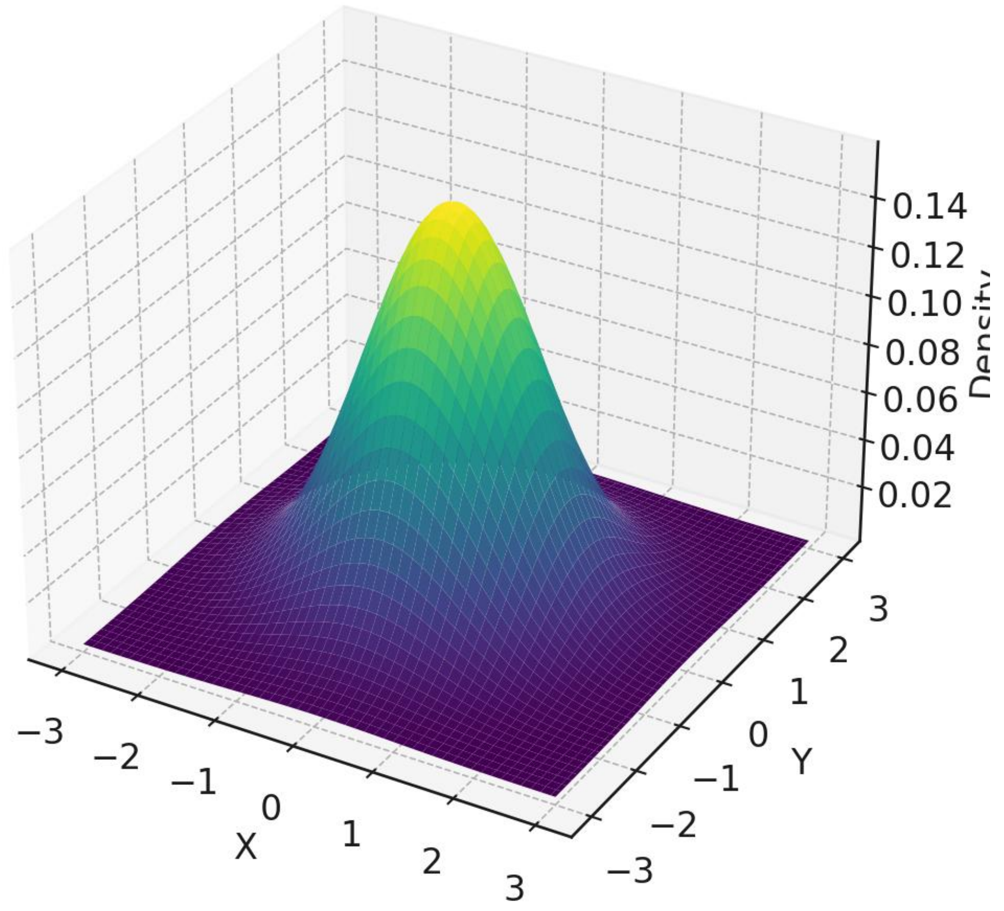


Also denoted by N(0, 1)

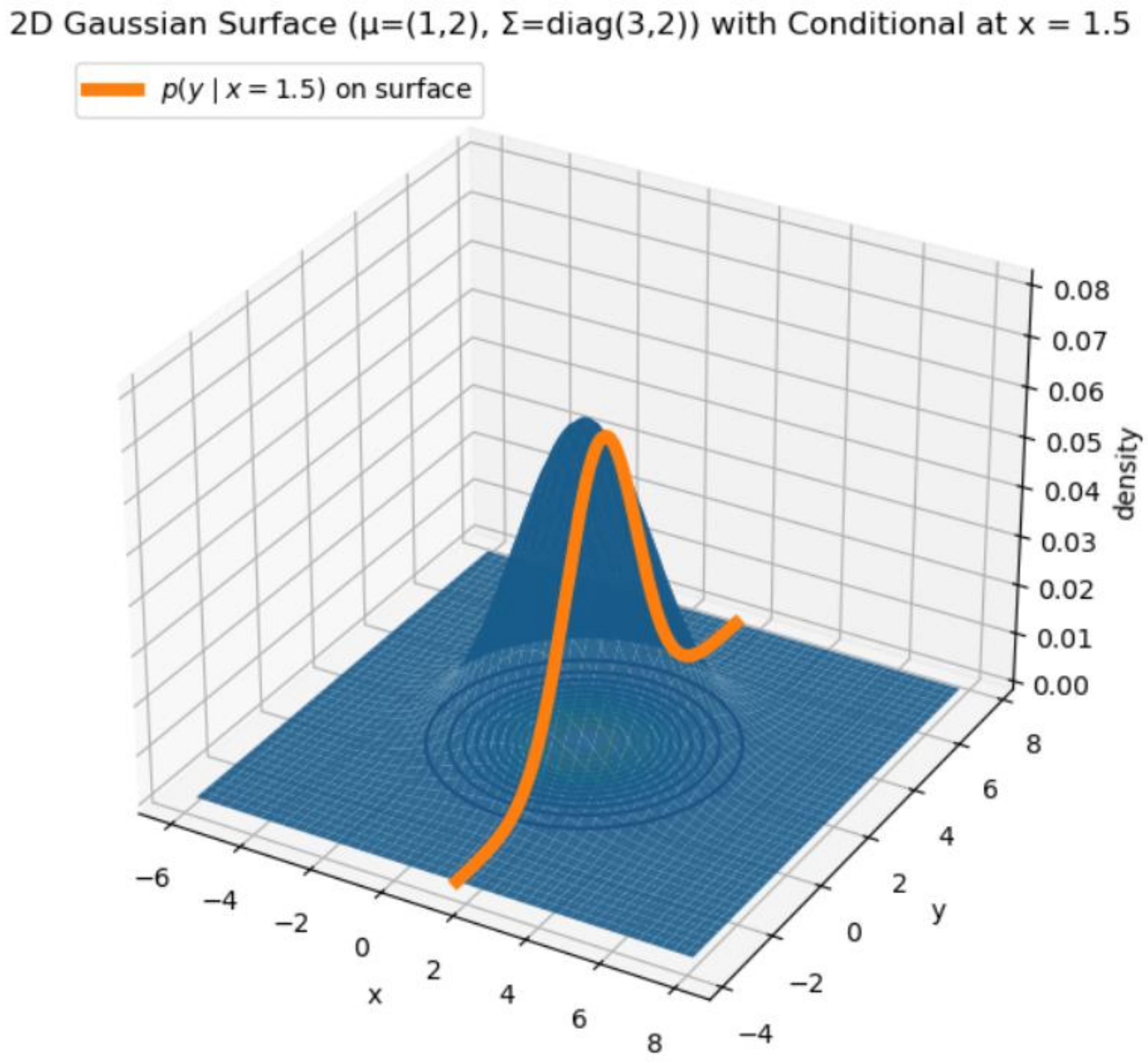
A Gaussian r.v. with mean m and Standard deviation s, is generated as:

g = m + s*torch.randn(…)

2D Gaussian PDF

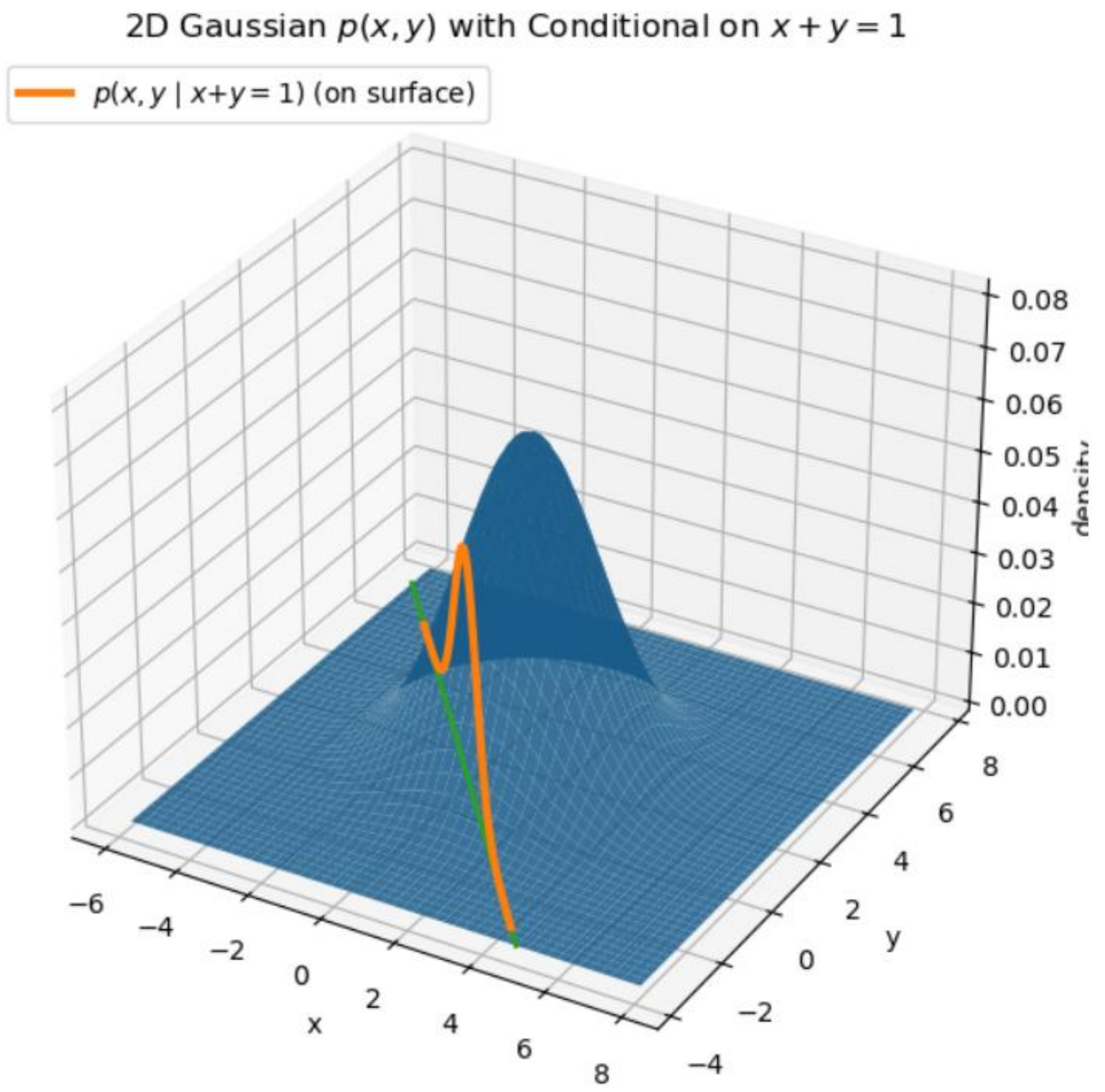


One more element: Conditional density



Conditions can be complicated…

$$p(x, y \mid \text{condition}) = \frac{p(x, y)}{p(\text{condition})},$$

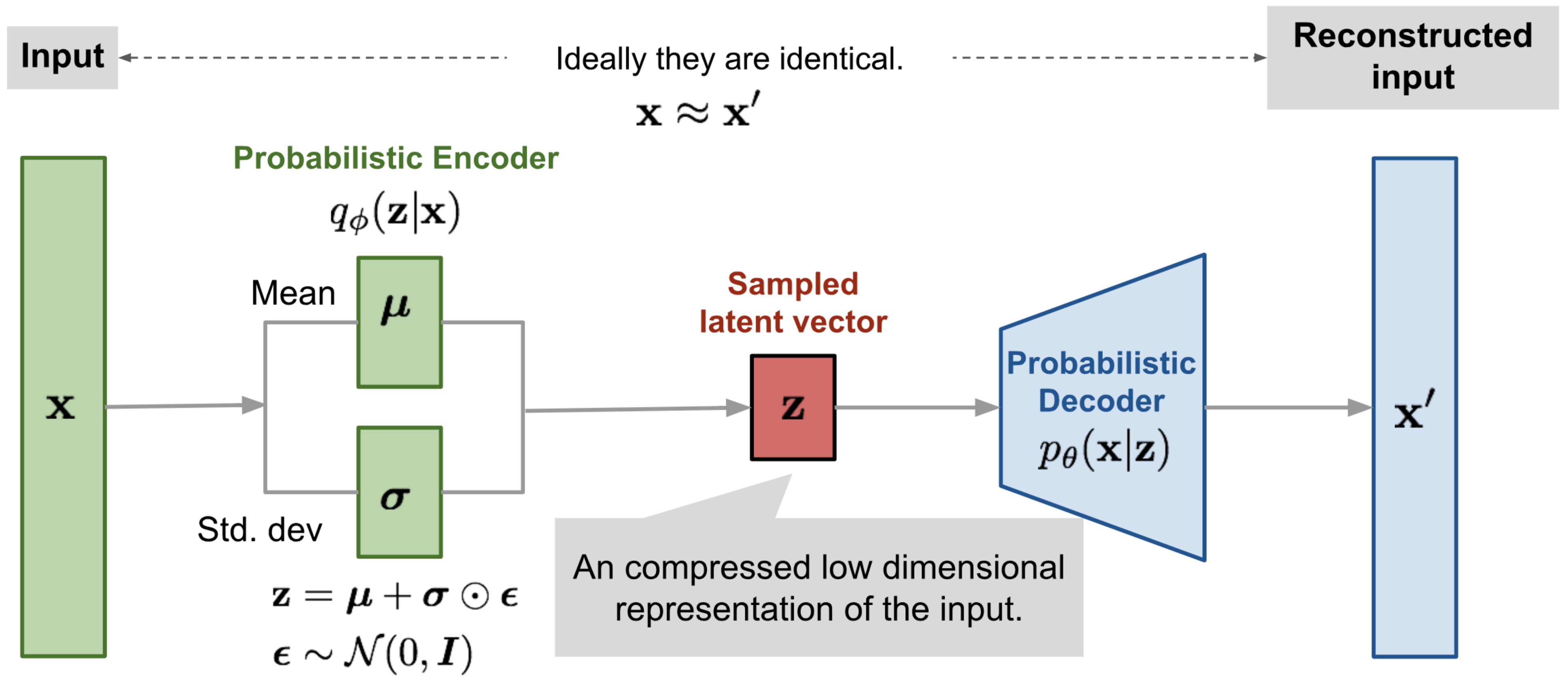


Just an aside: Can we generate the cameraman picture from its histogram?

No

We lost all spatial organization of pixels in the histogram

Variational autoencoder: Probabilistic encoder and decoder



Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
Probabilistic Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
$\mathbf{x}$
Mean
$\boldsymbol{\mu}$
Std. dev
$\boldsymbol{\sigma}$
$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$
Sampled latent vector
$\mathbf{z}$
An compressed low dimensional representation of the input.
Probabilistic Decoder
$p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$

https://lilianweng.github.io/posts/2018-08-12-vae/

Variational autoencoder…

We want: $q_\phi(z|x) \approx p_\theta(z|x)$

Probability of latent code from the encoder side given image x

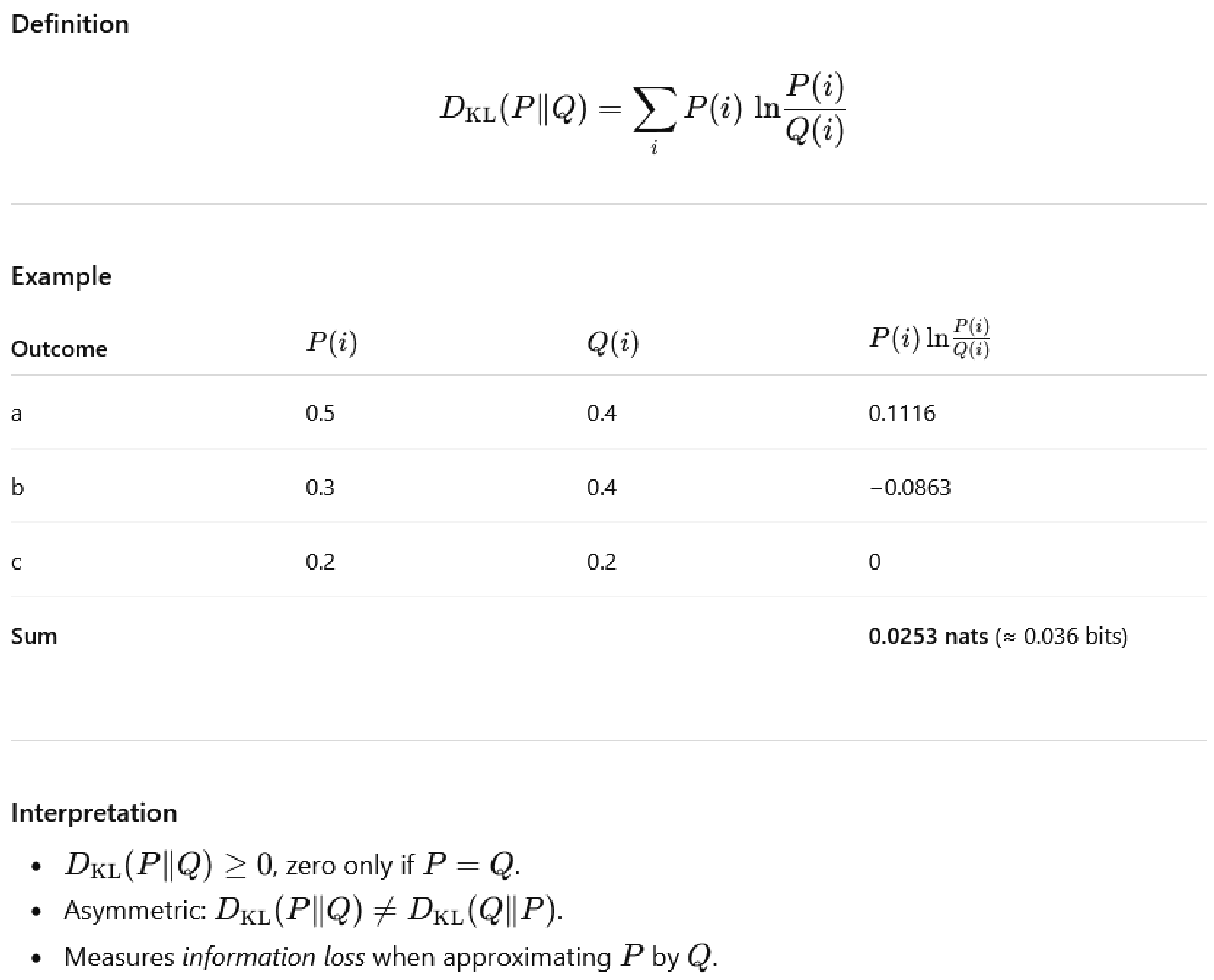
Probability of latent code from the decoder side given image x

After a ton of math (https://lilianweng.github.io/posts/2018-08-12-vae/), our loss function:

Image Reconstruction + KL-divergence between $q_\phi(z|x)$ and *N(0,1)*

KL-divergence: a type of distance between density functions

If you imagine encoding outcomes optimized for Q, but nature draws from P: $D_{\mathrm{KL}}(P \parallel Q)$ tells you the **expected number of extra bits (or nats)** per sample you waste.



Definition

$$D_{\mathrm{KL}}(P\|Q) = \sum_i P(i)\,\ln\frac{P(i)}{Q(i)}$$

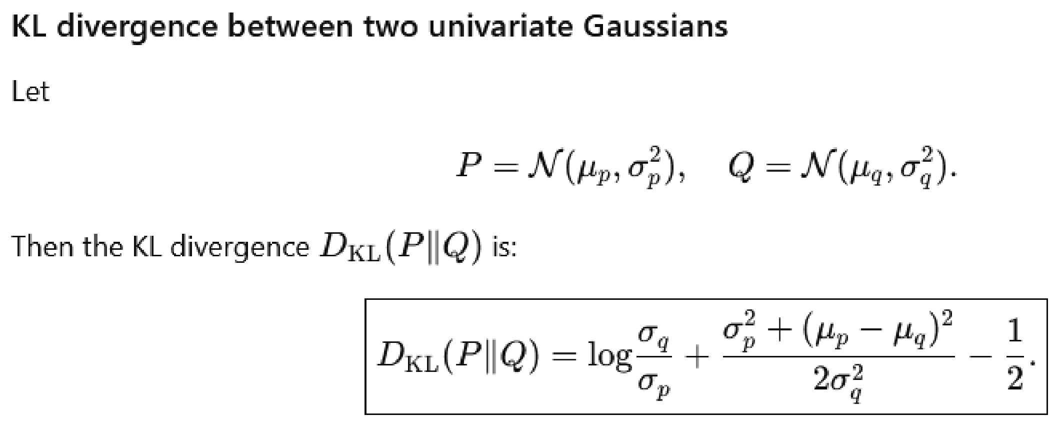
Example

Outcome	$P(i)$	$Q(i)$	$P(i)\ln\frac{P(i)}{Q(i)}$
a	0.5	0.4	0.1116
b	0.3	0.4	−0.0863
c	0.2	0.2	0
Sum			**0.0253 nats** (≈ 0.036 bits)

Interpretation

- $D_{\mathrm{KL}}(P\|Q) \geq 0$, zero only if $P = Q$.
- Asymmetric: $D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P)$.
- Measures *information loss* when approximating P by Q.

KL-divergence between two univariate Gaussians



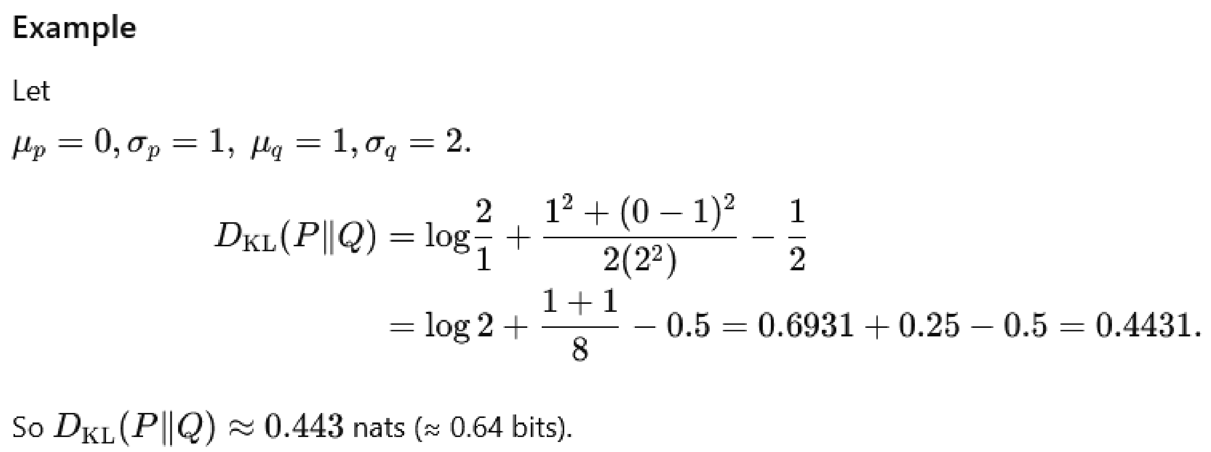
KL divergence between two univariate Gaussians

Let

$$P = \mathcal{N}(\mu_p, \sigma_p^2), \quad Q = \mathcal{N}(\mu_q, \sigma_q^2).$$

Then the KL divergence $D_{\mathrm{KL}}(P\|Q)$ is:

$$\boxed{D_{\mathrm{KL}}(P\|Q) = \log\frac{\sigma_q}{\sigma_p} + \frac{\sigma_p^2 + (\mu_p - \mu_q)^2}{2\sigma_q^2} - \frac{1}{2}.}$$



Example

Let

$\mu_p = 0, \sigma_p = 1,\ \mu_q = 1, \sigma_q = 2.$

$$\begin{aligned} D_{\mathrm{KL}}(P\|Q) &= \log\frac{2}{1} + \frac{1^2 + (0-1)^2}{2(2^2)} - \frac{1}{2} \\ &= \log 2 + \frac{1+1}{8} - 0.5 = 0.6931 + 0.25 - 0.5 = 0.4431. \end{aligned}$$

So $D_{\mathrm{KL}}(P\|Q) \approx 0.443$ nats (≈ 0.64 bits).

KL-divergence term in VAE

Setup:

$q_\phi(z|x) = \mathcal{N}(z;\, \mu_\phi(x),\, \mathrm{diag}(\sigma_\phi(x)^2)),$
$p(z) = \mathcal{N}(0, I)$

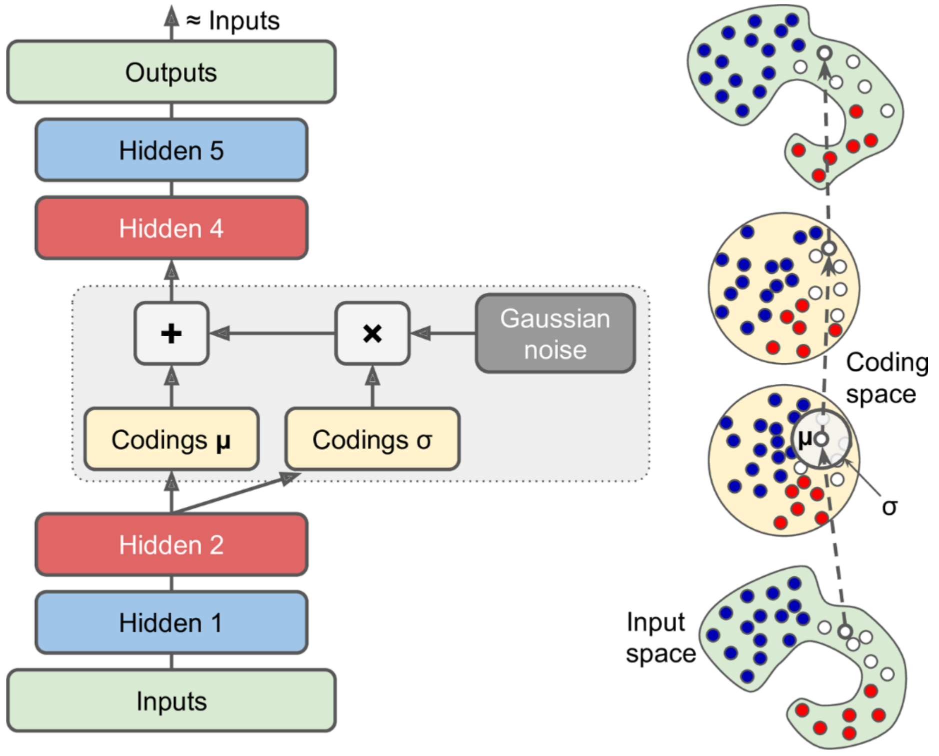
KL-divergence term:

$$D_{KL}(q_\phi(z|x) \,|\, p(z)) = \frac{1}{2}\sum_i (\mu_i^2 + \sigma_i^2 - \log \sigma_i^2 - 1)$$

PyTorch implementation:

```
kl = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
```

Variational autoencoder…



Cost function has two components:

Image Reconstruction +
KL-divergence (constraints on μ and σ)

The constraint: the encoded distribution should look like a zero-mean, unit variance Gaussian.

https://lilianweng.github.io/posts/2018-08-12-vae/

https://www.youtube.com/watch?v=OM95kDPAW-M

The advantage is that you can generate data (images) that look like the training images.

MNIST_Variational_AE.ipynb

Issues with VAE

1. Blurry Reconstructions

- VAEs average over many possible versions of an image.
- This produces smooth, washed-out results instead of sharp details.

2. Posterior Collapse

- The encoder sometimes stops using the input image.
- All latent codes become nearly identical, so the model ignores the latent space entirely.

3. Reconstruction vs. Regularization Trade-off

- Balancing image quality and latent structure is difficult.
- Too much regularization = poor reconstructions;
  too little = unstable or meaningless latent space.

4. Overly Smooth Latent Space

- The continuous Gaussian latent space blends different concepts together.
- Interpolations between unrelated images create unrealistic "hybrid" results.

5. Hard to Combine with Discrete Generative Models

- PixelCNNs and Transformers work best with discrete symbols.
- VAEs use continuous latents, so they can't take advantage of these powerful priors.

6. Loss of Fine Details

- The decoder models pixels independently, ignoring local texture patterns.
- This leads to loss of crisp edges, fine textures, and high-frequency detail.

VQ (Vector Quantized)-VAE

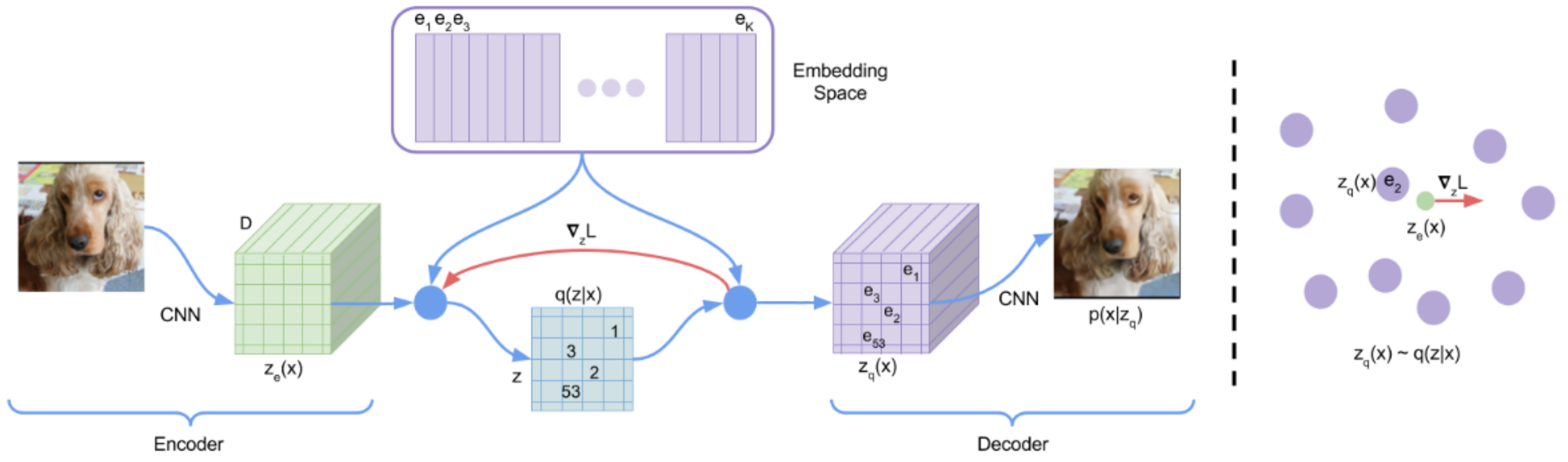
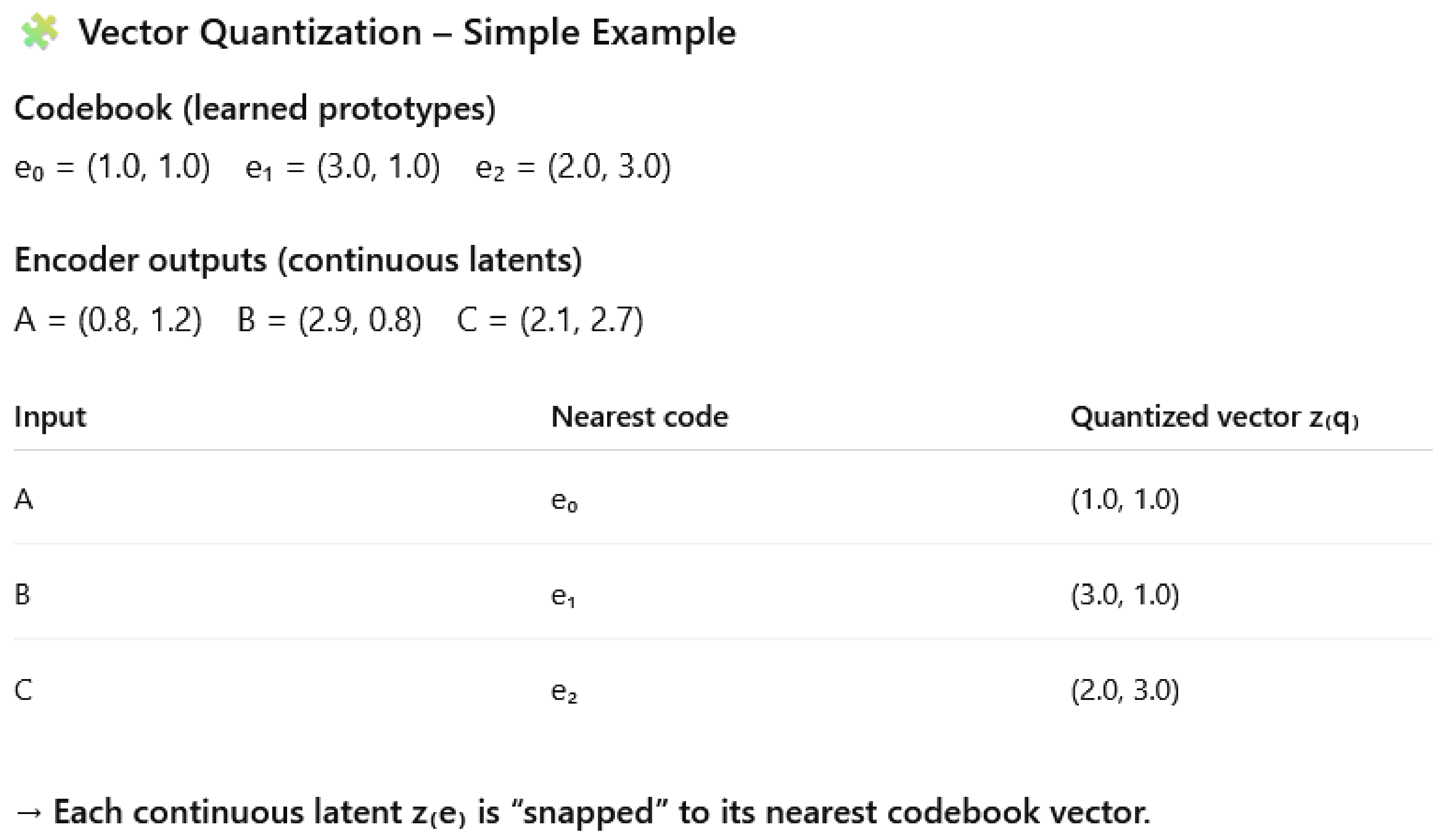


Figure 1: Left: A figure describing the VQ-VAE. Right: Visualisation of the embedding space. The output of the encoder $z(x)$ is mapped to the nearest point e_2. The gradient $\nabla_z L$ (in red) will push the encoder to change its output, which could alter the configuration in the next forward pass.

https://arxiv.org/pdf/1711.00937

Vector quantization: toy example



🧩 Vector Quantization – Simple Example

Codebook (learned prototypes)

$e_0 = (1.0, 1.0)$ $e_1 = (3.0, 1.0)$ $e_2 = (2.0, 3.0)$

Encoder outputs (continuous latents)

A = (0.8, 1.2) B = (2.9, 0.8) C = (2.1, 2.7)

Input	Nearest code	Quantized vector $z_{(q)}$
A	e_0	(1.0, 1.0)
B	e_1	(3.0, 1.0)
C	e_2	(2.0, 3.0)

→ Each continuous latent $z_{(e)}$ is “snapped” to its nearest codebook vector.

VQ-VAE…

🧩 1. The codebook

- Shape: **(K × D)**
  - `K` = number of code entries (e.g., 256)
  - `D` = dimension of each embedding vector (e.g., 64)

So each row in the codebook is a **prototype vector** of length `D`.

```
Codebook (K=4, D=2):

Index    Embedding vector
0        [ 1.2, -0.7 ]
1        [-0.3,  0.9 ]
2        [ 2.1,  1.5 ]
3        [ 0.0, -1.1 ]
```

🧠 2. The encoder output

When the encoder processes an image, it doesn't give *just one* D-dimensional vector — it gives a **spatial grid** of them.

- Shape: **(B × D × H × W)**
  - `B` = batch size
  - `D` = feature dimension (same as codebook embedding size)
  - `H × W` = latent grid size (e.g., 7×7 for 28×28 MNIST)

So each position in that grid (each of the `H×W` locations) is a **D-dimensional vector**. That's the one that gets quantized.

🔢 3. Quantization step

For every one of those `H×W` vectors:

1. Compute distance to all `K` codebook vectors (each `D`-dim).
2. Pick the nearest one.
3. Replace the encoder vector with that codebook embedding.

Result: a quantized latent map

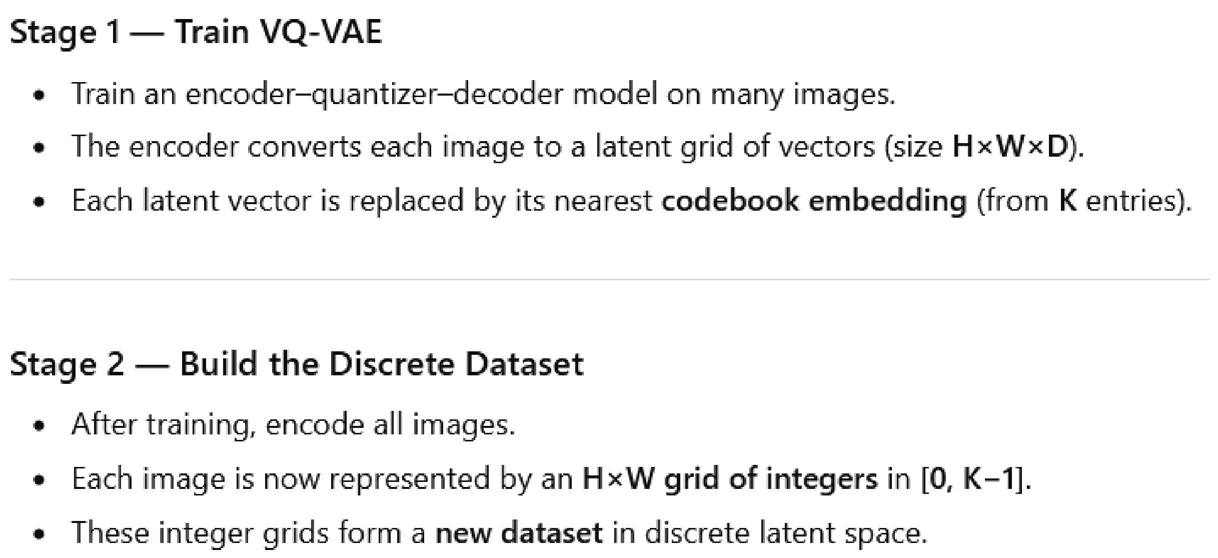
- shape **(B × D × H × W)** again (same shape as input)
- but with values *snapped* to codebook vectors.

🧩 4. Discrete view

You can also represent the quantized grid as:

- **indices** of shape `(B × H × W)` — each element ∈ `[0, K-1]`
  (which codebook row was chosen for that pixel in latent space)

VQ-VAE Image generation

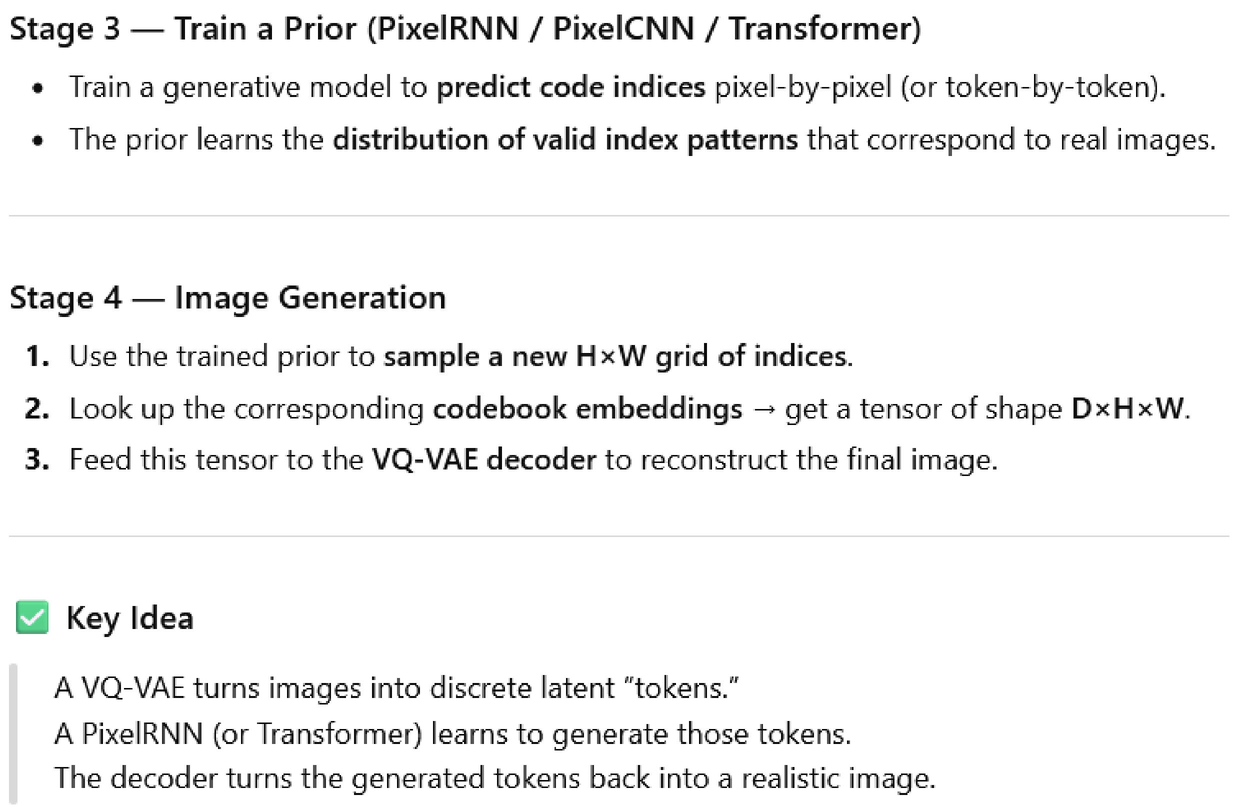


Stage 1 — Train VQ-VAE

- Train an encoder–quantizer–decoder model on many images.
- The encoder converts each image to a latent grid of vectors (size **H×W×D**).
- Each latent vector is replaced by its nearest **codebook embedding** (from **K** entries).

Stage 2 — Build the Discrete Dataset

- After training, encode all images.
- Each image is now represented by an **H×W grid of integers** in [**0, K−1**].
- These integer grids form a **new dataset** in discrete latent space.



Stage 3 — Train a Prior (PixelRNN / PixelCNN / Transformer)

- Train a generative model to **predict code indices** pixel-by-pixel (or token-by-token).
- The prior learns the **distribution of valid index patterns** that correspond to real images.

Stage 4 — Image Generation

1. Use the trained prior to **sample a new H×W grid of indices**.
2. Look up the corresponding **codebook embeddings** → get a tensor of shape **D×H×W**.
3. Feed this tensor to the **VQ-VAE decoder** to reconstruct the final image.

✅ **Key Idea**

> A VQ-VAE turns images into discrete latent "tokens."
> A PixelRNN (or Transformer) learns to generate those tokens.
> The decoder turns the generated tokens back into a realistic image.

Why VQ-VAE Works Better Than a VAE

🧠 1. Continuous vs. Discrete Latents

- A standard **VAE** uses a **continuous Gaussian latent space**.
  - Pros: smooth interpolation
  - Cons: tends to produce **blurry reconstructions** because it averages over many possible latent codes.
- A **VQ-VAE** uses **discrete codebook entries** (nearest prototypes).
  - Enforces *crisp decisions* → sharper reconstructions.
  - Reduces posterior variance — no sampling noise from Gaussians.

🎨 2. Sharper Reconstructions

- VAEs reconstruct images from *probabilistic* latents → decoder must handle uncertainty → blurred results.
- VQ-VAEs replace that uncertainty with **hard assignments** → decoder sees *clean, consistent codes* → produces sharper, more detailed images.

🧩 3. Discrete "Vocabulary" Enables Language-like Modeling

- The codebook gives a **discrete latent vocabulary** (like visual words).
- This allows powerful **autoregressive models** (PixelCNN, Transformer) to model image structure effectively —
  something that's hard in a continuous latent space.

🧮 4. No KL-collapse problem

- VAEs often suffer from *posterior collapse* (the encoder ignores the latent).
- VQ-VAE sidesteps this entirely — the quantization and commitment losses naturally keep encoder and codebook aligned.

🔉 5. Better compression & interpretability

- Discrete latents are easier to store, transmit, and interpret.
- The learned codebook can be seen as a "palette" of visual patterns or texture prototypes.